RDynamic is an **R** Package for Dynamic Modeling

R. Woodrow Setzer
National Center for Computational Toxicology
US Environmental Protection Agency

July 16, 2008

1 Introduction

Why another package for dynamic modeling? While other packages are quite good at facilitating dynamic modeling, they generally are fairly inadequate when it comes to using the resulting dynamic models with data. Real data come from often complex experimental designs, so that the least squares and weighted least squares approaches that are available in modeling packages that focus on dynamic modeling are often inadequate. **RDynamic** was written to allow dynamic models (models written as systems of ordinary differential equations) to be written in a simple syntax, then translated into a c source file that can be compiled, dynamically loaded, and integrated with the **deSolve** package already available in **R**.

While it may prove useful in other arenas, **RDynamic** was written primarily to help with physiologically-based pharmacokinetic and pharmacodynamic models. Such models typically are defined as systems of a few tens of ordinary differential equations, and may have dozens or hundreds of parameters, most of which are based on known (or supposed) physiological constants. Unlike many dynamic models in biomathematics, whose value is largely in their qualitative results, PBPK/PD models are intended to produce quantitative predictions. Thus, there is value in being able to develop such models so that they can be embedded in sophisticated statistical methodologies.

2 Using a Model Produced by **RDynamic**

Models produced by `RDynamic::ode2c()` are used just like any other **R** function. When it is called, the model function runs the dynamic model and returns the values of state variables at a prespecified set of desired time points. The result of such a call is often referred to as a "simulation" in what follows.

3 Components of a Dynamic Model

It is convenient to consider the variables in a dynamic model with respect to how their values change in the course of a simulation. **RDynamic** divides variables in a model into two groups. Parameters control the detailed behavior of a model, and are constant through any simulation. They may be set independently, or may depend upon the values of other parameters. Once set at the beginning of a simulation run, the values of all parameters remain constant. Variables are all the variables that change value during a simulation run. Together, they describe those aspects of a system that evolve during a simulation, and, thus, model those values of the target system that change over time. Variables include both state variables, whose value is dermined intrinsically, for example, as described by systems of difference or differential equations, and which require initial values, variables whose value is determined extrinsically (or forcings), and variables that are functions of the other variables. An example of the first in a PBPK model are amounts of parent compound in a tissue compartment. An example of a variable whose value is determined extrinsically would be an indicator variable that changes from 0 to 1 at 8:00 am and from 1 to 0 at 5:00 PM, and indicates whether an inhalation exposure is occurring. Finally, the concentration of parent compound is of interest, and is the amount divided by the tissue volume, in a PBPK model.

The model definition needs to describe several characteristics of the dynamic system:

- the values of parameters, and how the values of some parameters are derived from the values of others;
- the initial values of state variables (which may be determined by parameters);
- and how each state variable changes value, as a function of time, parameter values, and the values of other state variables.

Two special cases for how state variables change value are covered in special structures in **RDynamic**: continuous change over time, described by systems of ordinary differential equations, and discrete changes over time, described by structures called events. The value of any given state variable may jump from time to time, or when state variables satisfy a predetermined condition, as well as change continuously between jumps. For example, (at least as an approximation) stomach contents change continuously in time as the stomach empties into the small intestine, but may jump discretely with periodic gavage dosing or eating events.

A simulation of an **RDynamic** model results in a matrix of values of state variables at a discrete set of times, specified in advance of the simulation (as well as auxiliary information about the run that produced the output, to facilitate documentation).

4 The **RDynamic** Language

In **RDynamic**,thecharacteristicsthattogetherdescribeadynamicsystemaredefinedinaseparateblockofcode,each with its own syntax. It is better to think of describing a dynamic system with **RDynamic**, rather than programming it. Thus, statements within each block may be written in any convenient order (with some logical exceptions in the EVENT block). In particular, values may be used before they are defined, if that leads to a clearer exposition. The functionOde2ctakescareofturningthemodeldescriptionintoaprogramthatcanbeexecutedtosimulatethedynamic system. Two kinds of comments are available to clarify the model exposition and facilitate the construction of online documentation for the model. Units may be specified and will be tracked if supplied. In this section, I first go through an example of a model in **RDynamic**, then describe each of the elements of the language in more detail.

4.1 Example: PBPK Model for Pyrethroid Kinetics

This example is included in full in the appendix.

4.1.1 Prolog

RDynamic includes language elements designed to encourage model documentation, by allowing narrative documentation in the model file that will be incorporated into online documentation in the final compiled package. The model prolog includes several subsections.

```
@Title: Model for Pyrethroid kinetics
@Version: 11
@Date 09/20/2007
@Author: R. Woodrow Setzer
@BEGIN Description
Model for pyrethroid absorption, distribution, and metabolism. The
parameter values in the current file are for deltamethrin, with physiological
parameters for rats, but the intent is for the model structure to be the
```

```
same for any pyrethroid, and humans as well as rats.
@END Description
@BEGIN DOC
This is a translation of Matlab code by Mirfazaelian et al., modified by
Rogelio Tornero and Steve Godin.  The translation closely follows the
original code; the ODEs
are largely in the same order as in the original.  State variables are
slightly reordered.
@END Doc
```

Note the keyword:value pairs for keywords Title, Version, Date, and Author. They are introduced with an "@", and separated by ":" and any number of spaces. The values in each case should be short, and are incorporated both into the online help and the model object itself. In particular, the version number is saved with the compiled model, and documents results, so any simulation output contains the version number for the model that produced it. Thus, as long as the modeler is careful to update the version number of each model, it is possible to track modeling results back to the version of the model that produced them. Two longer fields are available in the prolog: "Description" and "DOC". These are both introduced by "@BEGIN" and closed with "@END", followed by the name of the field. The "Description" field should be fairly short, just a few sentences that give a high level description of the model. Any more detailed documentation should go into "DOC".

4.1.2 **VARIABLES**

The VARIABLES block contains declarations and (when required) initializations of quantities that change with time. These variables fall into three general categories: state variables, which need to be assigned initial values, and whose trajectory through time define the dynamic system; inputs, whose values change autonomously through time, and represent forcings for the dynamic system (such as repeated dosing in a pharmacokinetic model, periodic nutrient inputs in an ecological model, or greenhouse gas concentrations in a climate model); and variables whose value is interesting, but which are functions of other values in the system (such as concentrations, when the corresponding masses are state variables). These latter variables do not need initialization, but do need to be documented, and flagged so their values are retained. VARIABLES contains two sub-blocks for declaring state variables and inputs, each bracketed with a BEGIN – END pair.

Any state variable in the CONTINUOUS or JUMPS portion of the model whose value is not just a function of other variables will need to be both declared and given initial values. This occurs in the STATE subblock. Initial values may either be constants (since this is a PBPK model for an exogenous compound, the initial concentration in all tissue compartments is 0.0, for example), or an algebraic expression involving the values of parameters (defined in the PARAMETERS block) and constants. Here is part of the VARIABLES block of the pyrethroid model (omissions are marked by elipses):

```
BEGIN VARIABLES
  BEGIN STATE # Names and initial values
...
    # Dosing  - Oral route
    ASTM    = stomach @ (umole) Amt in stomach ;
    AINT    = 0.0      @ (umole) Amt in intestine ;
    Oral    = 0.0      @ Amt absorbed via the oral pathway;
...
    # Dosing - IV route
    rIV = ivdose*BW*mol/Tinf @ rate of injection;
...

  END # State
  UER   @ Rate of urinary elimination;
  CA     @Concentration in arterial blood;
  CVF;
END #Variables
```

The block begins with "BEGIN VARIABLES" and ends with "END" (the "# State" is a comment which is not required, but helps make the code more readable). Each state variable that requires an initial value must appear in the STATE block, separated from its initial value by the "=" sign. The initial value may be numeric (e.g., all the 0.0 values) or an algebraic expression using variables defined in the PARAMETERS block. Document strings start with an ampersand ("@") and continue to the end of the line. Document strings should be thought of as optional parts of the statement. Statements are terminated with semicolons (";").

Variables that are simply functions of other variables, and whose values are of interest, are declared in a KEEP declaration. KEEP declarations begin with the keyword KEEP, followed by variable names and documentation strings, separated by commas. The entire statement is terminated by a semicolon. Only one KEEP statement is allowed in the STATE block, but it may span multiple lines. Note that, since the comma is used as a separator, it may not appear in any documentation strings in a KEEP statement. State variables may also be declared and initialized as arrays. See section 4.2.4 for details.

4.1.3 Comments

Two kinds of comments are used here and elsewhere in **RDynamic**. Comments that start with a "#" symbol are used to annotate the file containing the model description. The parser stops reading a line when it reaches a "#", so anything can be put after that symbol. This kind of comment is a good way to label different parts of the code. In the example, state variables are grouped and identified as, for example, relating to dosing or blood concentrations. Multiline comments can be created using the C-language style "/* */":

```
/* This is an example of a multi-line comment in RDynamic.
   It helps to indent the block on the left, so it is easier to read.
   Also, indent bullets:

   - Good for notating changes to the code

   - listing reasons for a particular construction

   - etc.

*/
```

The comment delimiters ("/* */") can appear anywhere on a line. The parser will ignore everything between them. Comments that start with a "@" introduce information that will go into the documentation file. In the STATE block, it is best to think of the declarations as having two separators: "=", that indicates an association between a state variable and its initial value, and "@", that separates the previous pair and a descriptor. This is a one-line comment, used generally for labeling a state variable. It is also is useful for documenting the units of the state variable.

4.1.4 **PARAMETERS**

The PARAMETERS block is syntactically similar to the STATE block. It is a list of variable names and initializers, separated by "=" signs. The "@" comment works just as it does in the STATE block, as do the other comments. Here is an excerpt from the pyrethroid model:

```
BEGIN PARAMETERS
# Dose-related ---------------------------------------------------
# Oral gavage:

oraldose = 0.0;
stomach = oraldose * BW * mol;

# IV Injection:
ivdose=0.0;
Tinf=0.005;

...

# Physiological Parameters (These are rat values) -------------------

BW      = 0.41                  @ kg;
QCC     = 14.10                 @ (L/h/kg^0.75) Brown et al [60 * 0.235];

...

# Deltamethrin specific parameters --------------------------------

MW = 505.                       @Molecular weight (ug/umol);
mol= 1000/MW                    @ correction factor for mg-->umol;

...

# Liver metabolic clearance
CLox      = 5.3                 @ L/h/kg;
CLest     = 0.0                 @ L/h/kg;
Kbld      = 0.0012              @ L/h/ml serum;

...

END # Parameters
```

Note that values on the right hand side of assignments may be numbers or algebraic expressions of other parameters. Users may assign new values at the beginning of a simulation to parameters that have been initialized with numbers (primary parameters). Parameters initialized with expressions (secondary parameters), are determined by the values of primary parameters and other secondary parameters. Their value can not be assigned arbitrarily in simulation runs.

As in other blocks, the modeler is free to write parameter definitions in any order that is convenient. This allows definitions of related parameters to be grouped together to facilitate documentation.

The algebraic expressions that appear on the right hand side of parameter declarations may include any of a large number of special functions. See section 4.2.2 for a list.

Parameters may also be declared and used as arrays. See 4.2.4 for more details.

4.1.5 **CONTINUOUS**

CONTINUOUS includes definitions of variables and their time derivatives, and describes how variables change value continuously in time. This is in contrast to JUMPS (discussed in the next section), which describe saltations in the values of state variables. This is also where variables whose values come from explicit functions of other state variables, parameters, and time are defined. Again, an excerpt from the pyrethroid model:

```
BEGIN CONTINUOUS
```

```
# Define concentrations based on state variable (amounts)
# I intracellular
# E extracellular

CA       = ABL/VBL;
...
# Concentration in the blood compartmetnt and blood clearance

CV     =  (QF*CVF + QR*CVR + QS*CVS + QBRN*CVBRN + QL*CVL)/QC;

CaEP' = Kbld*VBL*CA;        # blood clearace via esterases (umol/h)
ABL' = QC*CV - QC*CA - CaEP' + rIV;
...

# diffusion-limited compartments

AEF'   =  QF*(CA-CVF) + PAF*(CIF/PF-CVF);        # (umol/hr)
AIF'   =  PAF*(CVF-CIF/PF);
...
END #Continuous
```

There are no documentation strings in the CONTINUOUS section, but the other two comment types may be used freely. Note that derivatives are indicated by appending an apostrophe to state variable name (e.g. AEF'). Names of derivatives may be used on the right hand side of variable definitions, as well. Any variable that appears on the left hand side of an assignment that is not declared as a state variable will be invisible outside the scope of the CONTINUOUS block.

4.1.6 **JUMPS**

The JUMPS block defines when the values of state variables jump, and by how much. JUMPS must contain three sub-blocks to be complete:

- EVENT: defines what happens. Any state variable may appear on the right hand side of an assignment, and any legal expression involving state variables and parameters may appear on the left hand side. FOR loops and IF THENELSE control structures are allowed. The special variable TIME will contain the value of the time variable when the event was triggered. The special function INSERT_EVENT may be used to add new events to the action list (see ACTIONS, below. There may be and often will be multiple events defined in a model.
- TRIGGER: defines when events may happen. Generally, think of this as a function that returns 0.0 when you want to trigger an event. The final expression in a TRIGGER is the return value. It must be a scaler, and must be a legal right hand side expression. A trigger function may be a function of TIME and any or all state variables.

  A special form applies when you want to set a trigger to go off at a particular time. This is the form shown in the example, in which the final expression in the trigger definition is of the form "TIME == value;". There may be multiple TRIGGER blocks in JUMPS.
- ACTIONS: associates triggers with events. It is just a string of pairs, first a trigger, followed by an event. If there are multiple events listed for a trigger, they are executed in the order in this list. There must be only one ACTIONS block in a program.

Subblocks of JUMPS consist of one of the keywords with only white space (spaces and tabs) before it on its line, followed by a left curly brace, one or several legal expressions (defined above), followed by a right curly brace on a line by itself.

The function INSERT_EVENT(what, when) inserts the event 'what' into the list of events to be executed when the trigger 'when' goes off. The event 'what' must be defined in the JUMPS section. 'when' must be a defined trigger or the special phrase 'TIME == value', where value is a constant or variable whose value is available within the EVENT function from which INSERT_EVENT has been called.

In EVENTs and TRIGGERs, any undeclared variable will be assumed to be local, and its value will disappear on exit from the function. Only state or local variables may appear on the left hand side of assignments in event definitions, but right hand sides may include parameters, state variables, constants, and TIME. They may also include any of the functions available in the other blocks. Finally, special IF and IF ... THEN ELSE ... blocks are available for defining events. Here is the JUMPS block from the pyrethroid model:

```
BEGIN JUMPS
EVENT stop-infusion {
rIV = 0.0;
}
TRIGGER when_stop-infusion {
TIME == Tinf;
}
ACTIONS {
when_stop-infusion stop-infusion;
}
END # Jumps
```

Again, only "#" and "/* */" style comments are available.

4.2 Additional Language Details

4.2.1 Reserved Words

The following are reserved words, and should not be used as variable names: BEGIN, END, PARAMETERS, STATE, CONTINUOUS, FUNCTION, ARRAY, JUMPS, EVENT, TRIGGER, ACTIONS, IF, THEN, ELSE, FOR, TIME, KEEP. Reserved words are written in all capital letters.

4.2.2 Arithmetic Operators and Mathematical Functions

The following operators may be used whenever mathamatical operations are allowed: the usual operators for addition, subtraction, multiplication, and division: +-*/; exponentiation:^ or **. Math functions that are available are:

Function	Definition
exp(x)	exponential: e^x
log(x)	natural logarithm $(x > 0)$
log10(x)	base 10 logarithm $(x > 0)$
sin(x)	trigonometric sine function (argument in radians)
cos(x)	trigonometric cosine function (argument in radians)
asin(x)	trigonometric arcsine function ($1 < x < 1$)
acos(x)	trigonometric arccosine function ($1 < x < 1$)
sqrt(x)	square root $(x \perp 0)$
sinh(x)	hyperbolic sine
asinh(x)	hyperbolic arcsine
cosh(x)	hyperbolic cosine
acosh(x)	hyperbolic arccosine
gammafn(x)	the Gamma function
lgammafn(x)	natural log of the Gamma function
beta(a, b)	The complete beta function
lbeta(a, b)	natural log of beta(a, b)
ifthenelse(a,b,c)	b if a is not 0, else c (not for CONTINUOUS block)

Other functions may be added to these as needed.

4.2.3 Variable and Function Names

All variable names begin with an alphabetic character, and may include letters and numerals. All names are case-sensitive.

4.2.4 Arrays

RDynamic provides a limited implementation of arrays for parameters and state variables. Arrays are indexed beginning with 1 (as in Fortran and R). Arrays are declared in either PARAMETERS or STATE, and referred to as in an R array. That is, the indices are enclosed in square brackets, and indices for the different dimensions separated by commas. To assign a value to an element of an array, say var1[1], or var2[2,3]:

```
var1[1] = a + b * TIME;
var2[2,3] = 42;
```

A construct is available to conveniently initialize arrays in STATE and PARAMETERS, and FOR loops are available in CONTINUOUS and JUMPS for simplifying using arrays in repetitive structures. Except for the index variable declared in a FOR loop, array indices must be integer constants.

Declaring Arrays State variables and parameters may be components of arrays. Arrays must be declared in either the STATE or PARAMETERS blocks using the ARRAY statement:

```
ARRAY var1[10], var2[3,6];
```

Here var1 and var2 may be parameters (in a PARAMETERS block) or state variables (in a STATE block). The dimensions must be constant integers, and an arbitrary number of dimensions is allowed. On declaration, all elements in an array are initialized to 0.0.

Initializing Arrays Array elements in STATE or PARAMETERS may be initialized one-at-a-time just as any other variable:

```
var[12] = 12.9
```

Alternatively, elements of an array may be block initialized:

```
ARRAY var[2,3], var2[5];
var2 = {1., 2., 3., 4., 5.};
var = {{1.0, 5.0, 2.1}
       {var2[1], 23*Tstab, 7.3}};
```

Initializers must follow the same rules as any other initializer; array elements in an array must not be used to initialize other elements in the same array, and dependencies among arrays must form acyclic chains. That is, the following would be illegal:

```
ARRAY var[2,3], var2[5];
var2 = {1., 2., 3., 4., var[1,3]};
var = {{1.0, 5.0, 2.1}
       {var2[1], 23*Tstab, 7.3}};
```

If two arrays have the same extent, the initialization may be completely implicit:

```
BEGIN PARAMETERS
ARRAY Init[6];
Init = {12., 0., 0., 0., 0., 0.};
...
END
BEGIN STATE
ARRAY Svar[6];
Svar = Init;
...
END
```

FOR loops FOR loops are used in CONTINUOUS and JUMPS to simplify repetitive expressions. They are actually unrolled during translation (not during execution, so limits of FOR loops must be integer constants. The syntax is:

```
FOR (Indexvar IN llim:ulim) {
expressions
}
```

The variable Indexvar must not be declared elsewhere, and llim and ulim must be numeric constants. FOR loops may be nested.

For example:

```
Svar[1]' = r[1] * Svar[1] - (a[1] + m[1])*Svar[1];
FOR (I in 2:6) {
  Svar[I]' = a[I-1]*Svar[I-1] + r[I] * Svar[I] - (r[I] + m[I]) * Svar[I];
}
Svar[7] = a[6] * Svar[6] + r[7] * Svar[7] - m[7] * Svar[7];
```

4.2.5 Ordering and Separating Definitions in the **PARAMETERS** and **STATE** Blocks

Multiple declarations can appear on the same line (though this should generally be avoided, as it makes reading and documenting the code more difficult), and value expressions may extend over onto multiple lines.

PARAMETERS There is no ordering requirement for statements in the PARAMETERS block. The translater will optionally sort the declarations so that no value is used before it is declared. It is an error to define a parameter value using other than numeric constants or expressions involving numeric constants and other parameters.

STATE There is no ordering requirement for statements in the STATE block, either. The translator will optionally sort the declarations so that no value is used before it is declared. The STATE block should include declarations for every variable that changes through time (either continuously or in saltations through having its value defined in the JUMPS block) There must be a state variable for each derivative defined in the CONTINUOUS block, at least.

4.2.6 **CONTINUOUS**

The base variable name of every primed variable must appear in STATE; other variables appearing on the left hand side of assignments in CONTINUOUS that have not been declared in STATE will be treated as local (and their values inaccessible outside of the derivative definitions). The special variable TIME refers to the time at which the derivatives are being computed, to allow for inhomogeneous systems of equations.

4.2.7 JUMPS

Syntax of the IF and IF THEN ELSE Statements An EVENT or TRIGGER definition may contain IF and IF THEN ELSE statements to allow for conditional computation of changes to state variable values. The syntax is similar to the corresponding expression in C, except for mandatory curly braces:

```
IF (condition) {assignments, possibly multi-line}
```

and

```
IF (condition) {
  assignments, possibly multi-line
} ELSE {
  alternative assignments, possibly multi-line
}
```

These are like the same clauses in C, and unlike in R, in that the IF... statements do not have a return value, but are purely control structures.

The 'condition' expression may use the usual numeric comparison operators: >, >=, <, <=, ==; and the logical operators ! (for NOT), & (for AND) and | (for OR). Use parentheses liberally for grouping, but the usual C language priorities apply. The conditional expression is triggered if 'condition' evaluates to be a non-zero quantity.

5 Using **RDynamic**

The result of translating an **RDynamic** source file is an **R** source package, and, optionally, an **R** binary package, ready to be installed. The package contains online documentation for the model, and all the code to run it. To compile the pyrethroid example if the code from Appendix A is in the file pyrethroid.ode, and install it in the folder C:\RModels, the following commands in R would suffice:

```
> library(RDynamic)
> Ode2c("pyrethroiod.ode", compile=TRUE)
> install.packages("pyrethroid.zip", lib="C:/RModels", repos=NULL)
```

To use the newly compiled model:

```
> library(pyrethroid, lib.loc="C:/RModels")
```

Then, to get help for the function pyrethroid(),

```
> ?pyrethroid
```

will return all the documentation originally entered into the model file, as well as details on how to call the function which includes the names of all the modifiable parameters and all the state variable names.

The full call to the new model function looks like:

```
pyrethroid(times, ystart, RTol, ATol, Parms, ...)
```

'times' is a vector of times at which values of the state variables should be output. 'ystart' is an optional vector of initial values. If it is omitted, the initialization in the model file is used. However, it may be useful to continue the simulation for longer times after an initial run. Then, ystart is set to the final state in the previous model run, and times starts with the final time in the previous run. 'RTol' and 'ATol' control precision of the numerical solution; you may need to experiment with their values. 'Parms' is a named vector of default parameter values. It will default to the values in the model file, but it may be convenient to have several sets of values, for example for different species. Finally, values for individual parameters may be provided as arguments, as IVDOSE = 10, for example.

[more on the structure of the output here]

A Example: A Complete PBPK Model for Pyrethroid Kinetics